The Journal of Supercomputing (2023) 79:8124–8147
https://doi.org/10.1007/s11227-022-05003-3

Micro-kernels for portable and efficient matrix multiplication in deep learning

Guillermo Alaejos[1] · **Adrián Castelló**[1] · **Héctor Martínez**[2] · **Pedro Alonso-Jordá**[1] · **Francisco D. Igual**[3] · **Enrique S. Quintana-Ortí**[1]

Accepted: 4 December 2022 / Published online: 14 December 2022

Abstract
We provide a practical demonstration that it is possible to systematically generate a variety of high-performance micro-kernels for the general matrix multiplication (GEMM) via generic templates which can be easily customized to different processor architectures and micro-kernel dimensions. These generic templates employ vector intrinsics to exploit the SIMD (single instruction, multiple data) units in current general-purpose processors and, for the particular type of GEMM problems encountered in deep learning, deliver a floating-point throughput rate on par with or even higher than that obtained with conventional, carefully tuned implementations of GEMM in current linear algebra libraries (e.g., BLIS, AMD AOCL, ARMPL). Our work exposes the structure of the template-based micro-kernels for ARM Neon (128-bit SIMD), ARM SVE (variable-length SIMD) and Intel AVX512 (512-bit SIMD), showing considerable performance for an NVIDIA Carmel processor (ARM Neon), a Fujitsu A64FX processor (ARM SVE) and on an AMD EPYC 7282 processor (256-bit SIMD).

Keywords Matrix multiplication · Linear algebra libraries · High performance · Vector intrinsics · SIMD units

1 Introduction

A few decades ago, the basic linear algebra subprograms (BLAS) [1] sparked portability in scientific computing by defining a standard interface that hardware vendors could instantiate into their products, and researchers could then leverage from their codes and libraries. A significant leap forward toward combining portability with performance came later, in a seminal paper from Kazushige Goto and Robert A. van de Geijn in 2008 [2]. There, the authors define the foundations for the realization

✉ Adrián Castelló
adcastel@disca.upv.es

Extended author information available on the last page of the article

of the high performance matrix multiplication (GEMM) that underlies most current linear algebra libraries, such as GotoBLAS [2], BLIS [3], OpenBLAS [4], AMD AOCL, and (possibly) Intel MKL and oneAPI.

The ideas behind Goto and van de Geijn's algorithm for GEMM were eventually extended to formulate a rich family of algorithms for this operation that comprises six algorithms and three types of micro-kernels [5–7]. In [7], we investigated the members of this family, manually instantiating them on an NVIDIA Carmel, ARM-based processor, and conducting an evaluation for the type of operators that are encountered in convolutional neural networks [8, 9], *using a simple micro-kernel only*.

In this paper, we continue our previous work in [7, 9] toward improving portability (and maintainability) of GEMM. The key in this paper lies in the formulation of a common generic micro-kernel, which can then be easily instantiated into a collection architecture-specific realizations via a few macros and two configuration variables specifying the micro-kernel dimensions. This differs from current realizations of GEMM in libraries such as GotoBLAS2, OpenBLAS and BLIS, which integrate a single micro-kernel per architecture, encoded in assembly. This work thus makes the following specific contributions:

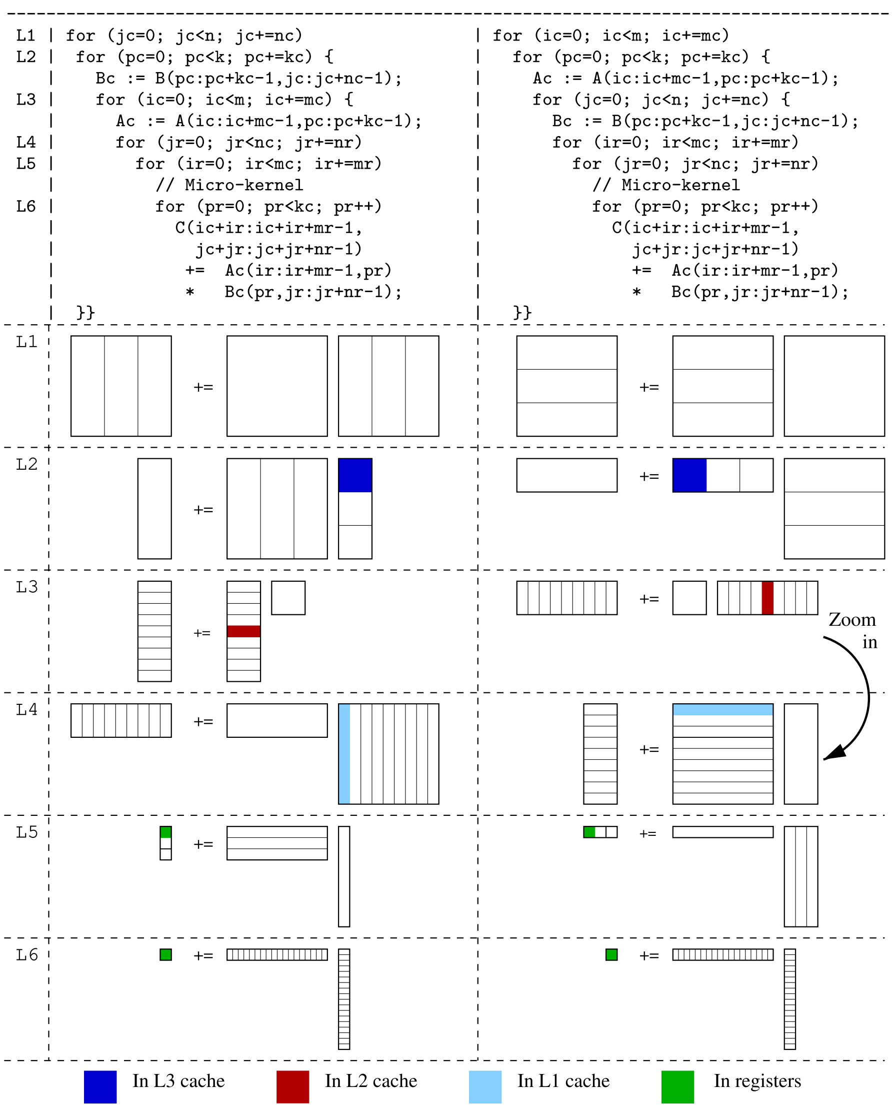
- We describe how to develop an ample variety of multi-platform, high performance micro-kernels for matrix multiplication using high level vector intrinsics to exploit the SIMD (single instruction, multiple data) units in current general-purpose processors.
- We integrate the intrinsics-based micro-kernels into the BLIS family of algorithms for matrix multiplication, experimentally exploring the performance of the family members.
- We provide practical evidence of the advantages and caveats of this high-level approach, for deep learning (DL) applications, on two ARM-based multicore processors, equipped with 128-bit and 512-bit SIMD units.

The rest of the paper is structured as follows. Sect. 2 briefly reviews the family of BLIS algorithms for high performance matrix-matrix multiplication. Sect. 3 explains how to generate GEMM micro-kernels using vector intrinsics. Sect. 4 is devoted to the experimental results. Finally, Sect. 5 summarizes the main results, presents some conclusions, and discusses future work.

2 The family of BLIS algorithms for GEMM

2.1 High performance implementation of GEMM

Consider the GEMM $C = C + AB$, where the operands are matrices with the following dimensions: $A \rightarrow m \times k$, $B \rightarrow k \times n$, and $C \rightarrow m \times n$. The BLIS framework (as well as other libraries, such as AMD ACML, OpenBLAS and ARMPL) follows GotoBLAS to encode this operation as three nested loops around two *packing routines* and a macro-kernel. In BLIS, the macro-kernel is decomposed into two additional



```
L1 | for (jc=0; jc<n; jc+=nc)                 | for (ic=0; ic<m; ic+=mc)
L2 |  for (pc=0; pc<k; pc+=kc) {              |   for (pc=0; pc<k; pc+=kc) {
   |    Bc := B(pc:pc+kc-1,jc:jc+nc-1);       |     Ac := A(ic:ic+mc-1,pc:pc+kc-1);
L3 |    for (ic=0; ic<m; ic+=mc) {            |     for (jc=0; jc<n; jc+=nc) {
   |      Ac := A(ic:ic+mc-1,pc:pc+kc-1);     |       Bc := B(pc:pc+kc-1,jc:jc+nc-1);
L4 |      for (jr=0; jr<nc; jr+=nr)           |       for (ir=0; ir<mc; ir+=mr)
L5 |        for (ir=0; ir<mc; ir+=mr)         |         for (jr=0; jr<nc; jr+=nr)
   |          // Micro-kernel                 |           // Micro-kernel
L6 |          for (pr=0; pr<kc; pr++)         |           for (pr=0; pr<kc; pr++)
   |            C(ic+ir:ic+ir+mr-1,           |             C(ic+ir:ic+ir+mr-1,
   |              jc+jr:jc+jr+nr-1)           |               jc+jr:jc+jr+nr-1)
   |             +=  Ac(ir:ir+mr-1,pr)        |               +=  Ac(ir:ir+mr-1,pr)
   |             *   Bc(pr,jr:jr+nr-1);       |               *   Bc(pr,jr:jr+nr-1);
   |  }}                                      |   }}
```

Fig. 1 Variants of the BLIS family for GEMM with C streamed from memory into the processor registers: B3A2C0 (left) and A3B2C0 (right)

loops around a *micro-kernel*, with the latter comprising a single loop that performs an outer product per iteration. The top-left part of Fig. 1 displays the *BLIS baseline algorithm* for GEMM, comprising the six loops, the two packing routines, and the micro-kernel.

Hereafter, we will refer to the BLIS baseline algorithm as B3A2C0. In this notation, $Z \in \{\text{A,B,C}\}$ specifies one of the three matrix operands and the subsequent number, $i \in \{0, 2, 3\}$, indicates the cache level where that operand resides (with 0

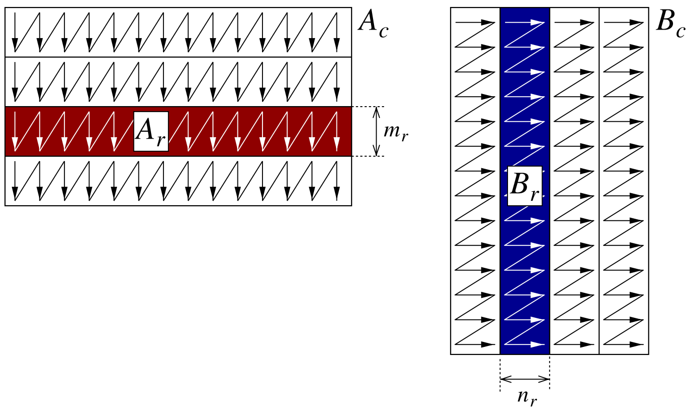


Fig. 2 Packing in the BLIS algorithm. The buffer A_c is packed into micro-panels of m_r rows while the buffer B_c is packed into micro-panels of n_r columns. When convenient, we will refer to these micro-panels as A_r and B_r, respectively

referring to the processor registers). The same matrix operand resides in both the L1 and L3 caches.

The three outermost loops of the B3A2C0 algorithm, indexed by j_c, p_c and i_c, respectively, traverse the n–, k– and m–dimension of the problem, partitioning the matrix operands conformally to the processor cache hierarchy. This nesting of the loops, together with the specific packing of B, A, and a selection of the cache configuration parameters n_c, k_c m_c, adjusted to the target processor architecture [10], favor that B_c remains in the L3 cache and A_c in the L2 cache during the execution of the micro-kernel, while C is streamed from the main memory into the processor registers; see the bottom-left part of Fig. 1 [10]. Besides, for high performance, the packing routines ensure that the contents of the buffers A_c, B_c are accessed with unit stride from the micro-kernel (see Fig. 2), and this specific part of the code is usually implemented using assembly in order to exploit the SIMD floating point units in current processors.

2.2 Meeting other members of the BLIS family

The B3A2C0 algorithm arranges the loops (from outermost to innermost) in the order $j_c \rightarrow p_c \rightarrow i_c \rightarrow j_r \rightarrow i_r \rightarrow p_r$, with the packing of B, A inserted within loops `L2`, `L3`, respectively; see Fig. 1.

Other variants of the BLIS family of algorithms for GEMM [5–7] can be obtained by nesting the loops differently, so as to favor that the matrix operands are "distributed" across the levels of the memory hierarchy following a different strategy. For example, as A, B are both input operands and play symmetric "roles" in the matrix multiplication, we can aim at maintaining A_c in the L3 cache, B_c in the L2 cache, while the micro-tile C_r resides in the processor registers, yielding the A3B2C0 variant. This can be done by reordering the loops of the BLIS baseline algorithm as $i_c \rightarrow p_c \rightarrow j_c \rightarrow i_r \rightarrow j_r \rightarrow p_r$ (that is, by swapping the loops for i_c,j_c and i_r,j_r) while

```
-------------------------------------------------------------------------------------------
L1 | for (jc=0; jc<n; jc+=nc)                | for (ic=0; ic<m; ic+=mc)
L2 |  for (pc=0; pc<k; pc+=kc) {             |   for (pc=0; pc<k; pc+=kc) {
   |    Bc := B(pc:pc+kc-1,jc:jc+nc-1);      |     Ac := A(ic:ic+mc-1,pc:pc+kc-1);
L3 |    for (ic=0; ic<m; ic+=mc) {           |     for (jc=0; jc<n; jc+=nc) {
   |      Cc := C(ic:ic+mc-1,jc:jc+nc-1);    |       Cc := C(ic:ic+mc-1,jc:jc+nc-1);
L4 |      for (pr=0; pr<kc; pr+=kr)          |       for (pr=0; pr<kc; pr+=kr)
L5 |        for (ir=0; ir<mc; ir+=mr)        |         for (jr=0; jr<nc; jr+=nr)
   |          // Micro-kernel                |           // Micro-kernel
L6 |          for (jr=0; jr<nc; jr++)        |           for (ir=0; ir<mr; ir++)
   |            Cc(ir:ir+mr-1,jr)            |             Cc(ir,jr:jr+nr-1)
   |              += A(ic+ir:ic+ir+mr-1,     |               +=  Ac(ir,pr:pr+kr-1)
   |                   pc+pr:pc+pr+kr-1)     |               *   B(pc+pr:pc+pr+kr-1,
   |              *  Bc(pr:pr+kr-1,jr);      |                     jc+jr:jc+jr+nr-1);
   |      C(ic:ic+mc-1,jc:jc+nc-1) := Cc;    |       C(ic:ic+mc-1,jc:jc+nc-1) := Cc;
   |  }}                                     |   }}
```

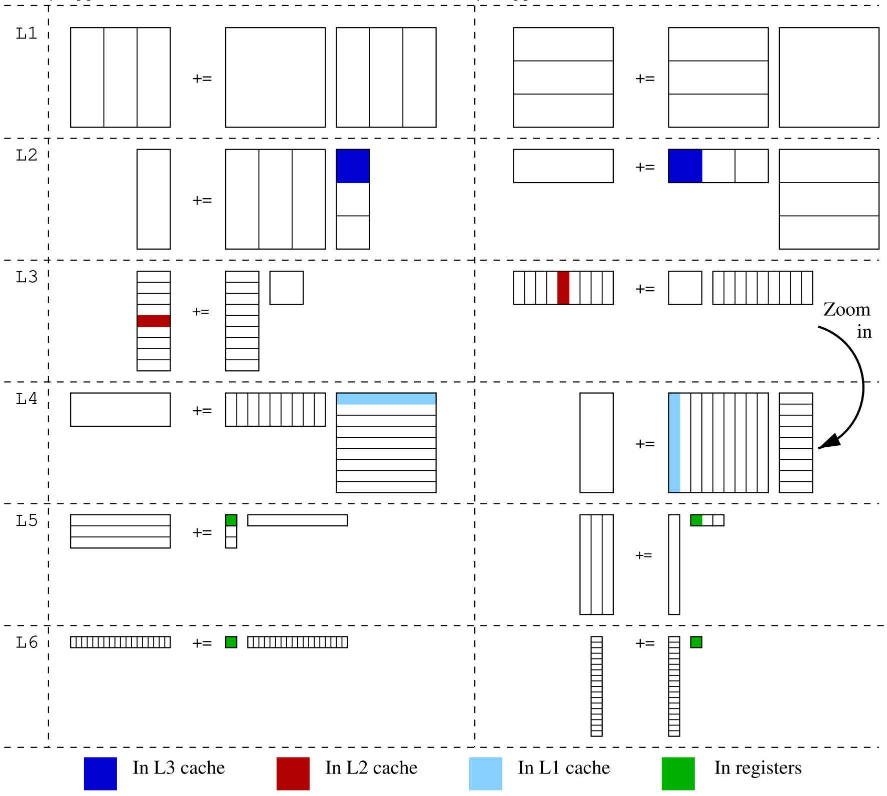


Fig. 3 Variants of the BLIS family for GEMM with C in the L2 cache: B3C2A0 (left) and A3C2B0 (right)

placing the packing of A, B within loops `L2`, `L3`, respectively (and choosing the appropriate values for m_c, n_c, k_c); see the right hand-side of Fig. 1.

The two variants previously described, B3A2C0 and A3B2C0, keep a micro-tile C_r in the processor registers, repeatedly updating its contents inside loop `L6` of the micro-kernel. The family of BLIS algorithms for GEMM can be enlarged by maintaining C in the L2 cache, yielding the two alternatives in Fig. 3. In the B3C2A0 variant (left hand-side of the figure), the loops are ordered as $j_c \rightarrow p_c \rightarrow i_c \rightarrow p_r \rightarrow i_r \rightarrow j_r$, and the two packing buffers contain blocks from B, C. With this solution, B_c resides in the L3 cache (and a part of it, in the L1 cache), while the micro-tile that is streamed from memory and remains in the processor registers, during the execution of the micro-kernel, corresponds to A. By swapping the roles of A and B in this variant, we obtain the A3C2B0 counterpart; see the right hand-side of Fig. 3.

Finally, the last two members of the BLIS family target the L3 cache with the C matrix: C3B2A0 and C3A2B0. In the first case, B_c is to remain in the L2 cache and a micro-tile A_r in the processor registers while in the second one, the roles of A and B are swapped. Again, this is attained with a specific ordering of the loops, (a proper selection of the cache configuration parameters, adjusted to the target processor architecture,) and the appropriate packing. For brevity, we do not provide the schemes of the last two algorithms, but they can be found in [7].

3 Micro-kernels for GEMM

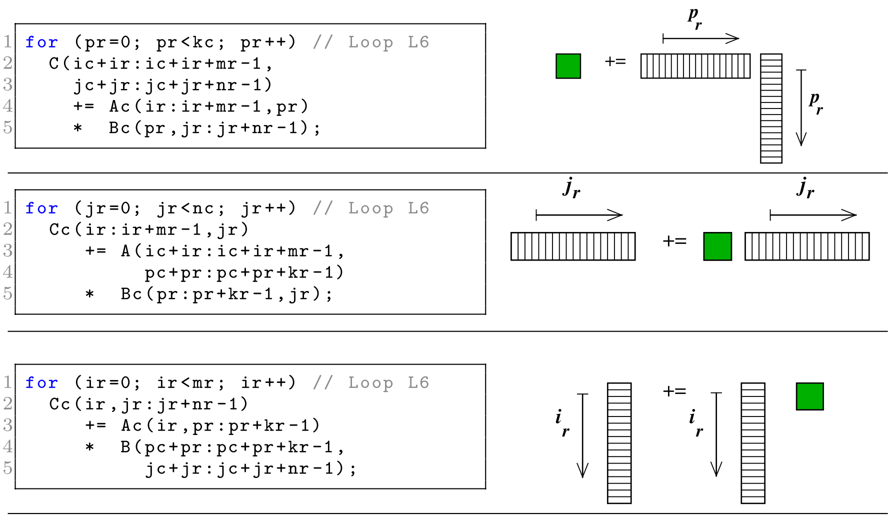
BLIS encodes the five outermost loops and the two packing routines in a portable, high level programming language such as C. In contrast, for high performance, BLIS integrates *architecture-specific micro-kernels, encoded using assembly*, for many processors from AMD, Intel, ARM and IBM. (Actually one micro-kernel per architecture.)

In this section, we depart from the BLIS conventional approach by exploring the use of high level micro-kernels, *totally encoded in the C programming language, that leverage vector intrinsics to exploit the SIMD units of the processor.* The goal is to enhance portability (and maintainability), possibly at the cost of some performance. (However, in the next section we will show that, our solution can also pay off in terms of performance, especially when we have to deal with DL applications on processors with 128-bit SIMD units.)

For simplicity, we choose 32-bit floating point (FP32) as the basic data type for all the routines/codes presented in the section (in C language, `float`), although the same ideas apply to other data types.

3.1 Quick overview of the BLIS micro-kernels

The family of BLIS algorithms for matrix multiplication relies on three types of kernels: The two variants with C resident in the processor registers (in Fig. 1) update a micro-tile of C via k_c outer products, as illustrated in Fig. 4 (top). In contrast, the variants that operate with A resident in the processor registers (e.g., left hand-side variant in Fig. 3) update a micro-tile of A via n_c matrix-vector products, as displayed in Fig. 4 (middle). Finally, the variants with B resident in the processor registers (e.g., right hand-side variant in Fig. 3) perform m_c vector-matrix products to update the micro-tile of B; see Fig. 4 (bottom). Hereafter we will refer to these three types of micro-kernels as *C-resident*, *A-resident*, and *B-resident* (to specify the matrix operand that stays in the processor registers).



```
for (pr=0; pr<kc; pr++) // Loop L6
  C(ic+ir:ic+ir+mr-1,
    jc+jr:jc+jr+nr-1)
    += Ac(ir:ir+mr-1,pr)
    *  Bc(pr,jr:jr+nr-1);
```

```
for (jr=0; jr<nc; jr++) // Loop L6
  Cc(ir:ir+mr-1,jr)
     += A(ic+ir:ic+ir+mr-1,
          pc+pr:pc+pr+kr-1)
     *  Bc(pr:pr+kr-1,jr);
```

```
for (ir=0; ir<mr; ir++) // Loop L6
  Cc(ir,jr:jr+nr-1)
     += Ac(ir,pr:pr+kr-1)
     *  B(pc+pr:pc+pr+kr-1,
          jc+jr:jc+jr+nr-1);
```

Fig. 4 Micro-kernels with C-resident (top), A-resident (middle) and B-resident (bottom)

Each type of micro-kernel requires a specific type of packing of two of the matrix operands. For the micro-kernel with C-resident, the packing schemes of A and B into buffers was already specified in Fig. 2; when the micro-kernel is A-resident, C_c, B_c are packed following the same pattern as A_c in Fig. 2 (though with B_c packed into blocks of k_r rows); and when the micro-kernel is B-resident, the buffers for C_c, A_c are packed as B_c in the same figure (with A_c packed in blocks of k_r columns).

3.2 An architecture-specific micro-kernel with *C*-resident for ARM Neon

We open the description of the high level micro-kernels by providing a simple example, with C-resident and $m_r \times n_r = 4 \times 4$ (dimension of the micro-tile of C), that leverages ARM Neon (intrinsics). We can highlight the following aspects in the routine in Listing 1:

- C is assumed to be stored by columns with leading dimension (that is, number of entries between two elements in the same row) `Clda`. The routine receives a pointer `C` into the appropriate entry of C.
- The buffers A_c, B_c are stored as displayed in Fig. 2. The routine receives pointers two `Ar`, `Br` into the appropriate entries of the respective buffers, corresponding to the "top-left entries" of the $m_r \times k_c$ and $k_c \times n_r$ blocks involved in the execution of the micro-kernel.
- ARM Neon operates with 128-bit vector registers (that is, 4 FP32 numbers per register). For ARMv8.2, there are 32 vector registers in total.
- Prior to the loop, four columns of C, each consisting of four FP32 numbers, C are loaded into four vector registers: `C0`–`C3` (Lines 15–16). After the loop, the contents of these vector registers are stored back into C (Lines 31–32).

This scheme is associated with the column-major layout of C in memory. For a matrix stored in row-major order, it would be more natural to load the rows of the micro-tile in the four vector register.

- At each iteration, the code loads one column of `Ar` into a vector register `ar` (Line 20), one row of `Br` into a vector register `br` (also Line 20), and updates `C0`–`C3` via four vector fused multiply-add intrinsics (Lines 23–26).

```
 1 gemm_ukernel_Cresident_Neon_4x4( int kc, float *Ar, float *Br,
 2                                   float *C,  int Clda ) {
 3 // 4x4 micro-kernel with C resident in regs.
 4 // Inputs:
 5 //   - C  stored in column-major order, with leading dimension Clda
 6 //   - Ar packed by columns, with leading dimension mr=4
 7 //   - Br packed by rows, with leading dimension nr=4
 8 
 9   int          pr, baseAB = 0;
10   float32x4_t C0, C1, C2, C3; // Columns of the 4x4 micro-tile of C
11   float32x4_t ar, br; // Single column/row of Ar/Br
12 
13   // Load the 4x4 micro-tile of C into the 4 vector registers,
14   // each with 4 FP32 elements
15   C0 = vld1q_f32(&C[0]);      C1 = vld1q_f32(&C[Clda]);
16   C2 = vld1q_f32(&C[2*Clda]); C3 = vld1q_f32(&C[3*Clda]);
17 
18   for ( pr=0; pr<kc; pr++ ) {
19     // Load the pr-th column/row of Ar/Br into vector registers
20     ar = vld1q_f32(&Ar[baseAB]); br = vld1q_f32(&Br[baseAB]);
21 
22     // Update the micro-tile with axpy
23     C0 = vfmaq_laneq_f32(C0, ar, br, 0);
24     C1 = vfmaq_laneq_f32(C1, ar, br, 1);
25     C2 = vfmaq_laneq_f32(C2, ar, br, 2);
26     C3 = vfmaq_laneq_f32(C3, ar, br, 3);
27     baseAB += 4;
28   }
29 
30   // Store the micro-tile in memory
31   vst1q_f32(&C[0],      C0); vst1q_f32(&C[Clda],   C1);
32   vst1q_f32(&C[2*Clda], C2); vst1q_f32(&C[3*Clda], C3);
33 }
```

Listing 1: Micro-kernel that operates with a 4×4 micro-tile and C-resident using ARM Neon intrinsics.

Encoding the micro-kernels with this level of abstraction, instead of using assembly, paves the road toward rapidly exploring many other versions. In this line, each iteration of the loop in the micro-kernel loads $m_r + n_r$ elements to perform $2m_rn_r$ floating point operations (flops). Choosing "squarish" micro-kernels (i.e., $m_r \approx n_r$) thus improves the ratio of flops to memory operations (also know as arithmetic intensity [11]). For the same reason, it is convenient to maximize the use of vector registers (without incurring into register spilling) [10]. For example, a 4×4 micro-kernel employs 6 vector registers (one for `Ar`, one for `Br`, and four for C) to deliver an arithmetic intensity of $32/8 = 4$. In comparison, an 8×12 micro-kernel employs 29 vector registers (two for `Ar`, 3 for `Br`, and $2{\cdot}12$ vector for C) to render significantly higher arithmetic intensity: $192/20 = 9.6$.

The micro-kernels can be further optimized by, on the one hand, integrating software pipelining [12] into the loop, in order to overlap computation and data transfers. For example, by using two additional vector registers, say `arn`, `brn`, at each iteration `jr`, we can initially load the `jr`+1 column/row of `Ar,Br` into them but operate with the data in `ar`, `br`. At the end of the iteration, in preparation for the next one, we copy the data from `arn`, `brn` to `ar`, `br`. On the other hand, loop unrolling [12] can help to reduce the overhead due to loop control. Finally, there exist vector intrinsics in many architectures to perform software prefetching that can significantly improve performance. In this work, we restrain ourselves from exploring this venue to focus on portability.

3.3 Comparison to micro-kernel with *A*-resident for ARM Neon

Developing an $m_r \times k_r = 4 \times 4$ micro-kernel with A-resident that leverages ARM Neon is direct, and results in the routine in Listing 2. A careful comparison of the micro-kernels with C-resident and A-resident reveals the following differences between them:

- For the same micro-kernel dimension (that is, when $m_r \times n_r = m_r \times k_r$), the variant with C-resident presents a higher arithmetic intensity. The reason is that, while both types of micro-kernels perform the same number of flops and number of loads from memory, the variant with A-resident has to write a column of C back into the memory at each iteration of the loop.
- The variant with A-resident repeatedly updates the vector register. This creates a RAW (read-after-write) dependency between the four vector fused multiply-add intrinsics (`vfmaq_laneq_f32`) that precludes their overlapped execution. An option to tackle this problem is to unroll the loop by a certain factor, but this comes at the cost of requiring a larger number of vector registers. This option constrains the dimensions of the micro-tile ($m_r \times k_r$) which do not produce register spilling.

```
gemm_ukernel_Aresident_Neon_4x4( int nc, float *Cr, float *Br,
                                         float *A,  int Alda ) {
// 4x4 micro-kernel with A resident in regs.
// Inputs:
//   - Cr packed by columns, with leading dimension mr=4
//   - A  stored in column-major order, with leading dimension Alda
//   - Br packed by rows, with leading dimension kr=4

  int         jr, baseCB = 0;
  float32x4_t A0, A1, A2, A3; // Columns of the 4x4 micro-tile of A
  float32x4_t cr, br; // Single column/row of Cr/Br

  // Load the 4x4 micro-tile of A into the 4 vector registers,
  // each with 4 FP32 elements
  A0 = vld1q_f32(&A[0]);       A1 = vld1q_f32(&A[Alda]);
  A2 = vld1q_f32(&A[2*Alda]); A3 = vld1q_f32(&A[3*Alda]);

  for ( jr=0; jr<nc; jr++ ) {
    // Load the jr-th columns of Cr/Br into vector registers
    cr = vld1q_f32(&Cr[baseCB]); br = vld1q_f32(&Br[baseCB]);

    // Update the jr-th column of C with axpy
    cr = vfmaq_laneq_f32(cr, A0, br, 0);
    cr = vfmaq_laneq_f32(cr, A1, br, 1);
    cr = vfmaq_laneq_f32(cr, A2, br, 2);
    cr = vfmaq_laneq_f32(cr, A3, br, 3);

    // Store jr-th column of Cr into memory
    vst1q_f32(&Cr[baseCB], C0);
    baseCB += 4;
  }
}
```

Listing 2: Micro-kernel that operates with a 4×4 micro-tile and A-resident using ARM Neon intrinsics.

3.4 A generic micro-kernel

We next take one significant step forward toward improving portability and maintainability by formulating a "generic" SIMD-enabled micro-kernel that is valid for any dimension $m_r \times n_r$. (For simplicity, we describe only the C-resident case, but the same ideas apply to the two other types of micro-kernels.) For this purpose, we rely on C macros to abstract the basic vector (data types and) intrinsics that were utilized earlier in the GEMM micro-kernels: load, store and AXPY (scalar α times x plus y) update. Furthermore, we will assume vector registers of b bits, each capable of storing `vl_fp32` $= b/32$ FP32 numbers. For an $m_r \times n_r$ micro-kernel with C-resident, this implies that we need $m_v \times n_r = (m_r/$`vl_fp32`$) \times n_r$ vector registers to store the micro-tile of C; m_v for the column of `Ar`; and $n_v = n_r/$`vl_fp32` for the row of `Br`. For simplicity, here we assume that m_r, n_r are both integer multiples of the vector register length `vl_fp32`.

```
 1 #define mv      (mr/vl_fp32)
 2 #define nv      (nr/vl_fp32)
 3
 4 gemm_ukernel_Cresident_SIMD_mrxnr( int kc, float *Ar, float *Br,
 5                                     float *C,  int Clda ) {
 6 // mr x nr micro-kernel with C resident in regs.
 7 // Inputs:
 8 //   - C  stored in column-major order, with leading dimension Clda
 9 //   - Ar packed by columns, with leading dimension mr
10 //   - Br packed by rows, with leading dimension nr
11
12   int        iv, j, jv, pr, baseA = 0, baseB = 0;
13   vregister Cr[mv][nr] // Micro-tile of C
14              ar[mv], br[nv]; // Single column/row of Ar/Br
15
16   vinit();
17
18   // Load the micro-tile of C into vector registers,
19   // each with vl_fp32 FP32 elements
20   for ( j=0; j<nr; j++ )
21     for ( iv=0; iv<mv; iv++ )
22       vload(Cr[iv][j], &C[j*Clda+iv*vl_fp32]);
23
24   for ( pr=0; pr<kc; pr++ ) {
25     // Load the pr-th column/row of Ar/Br into vector registers
26     for ( iv=0; iv<mv; iv++ )
27       vload(ar[iv], &Ar[baseA+iv*vl_fp32]);
28     for ( jv=0; jv<nv; jv++ )
29       vload(br[jv], &Br[baseB+jv*vl_fp32]);
30
31     // Update the micro-tile with axpy
32     for ( iv=0; iv<mv; iv++ )
33       for ( jv=0; jv<nv; jv++ )
34         for ( j=0; j<vl_fp32; j++ )
35           vupdate(Cr[iv][jv*vl_fp32+j], ar[iv], br[jv], j);
36     baseA += mr; baseB += nr;
37   }
38
39   // Store the micro-tile in memory
40   for ( j=0; j<nr; j++ )
41     for ( iv=0; iv<mv; iv++ )
42       vstore(&C[j*Clda+iv*vl_fp32], Cr[iv][j]);
43 }
```

Listing 3: Micro-kernel that operates with an $m_r \times n_r$ micro-tile and C-resident.

Listing 3 presents the generic micro-kernel, where we can highlight a couple of details:

- Prior to the main loop, indexed by `pr` (Line 24), we load the contents of the corresponding micro-tile of C into the array of vector registers `Cr` via two nested loops (Lines 20–22). The opposite transfer, from the vector registers back to memory, is carried out after the main loop (Lines 40–42).
- At each iteration of the main loop, a column of m_r elements of `Ar` and a row of n_r elements of `Br` are first loaded into the appropriate vector registers (Lines 26–27 and 28–29, respectively). These elements then participate in the update of the micro-tile stored in `Cr` Lines 32–35).

In practice, for high performance it is convenient to fully unroll the loops iterating over the m_r, n_r dimensions of the problem. (That is, those indexed by `iv`, `jv`, and `j`.) This can be done either with assistance from the compiler, manually, or automatically with a simple script generator (as described later in this section).

3.5 Adapting the generic micro-kernel to ARM Neon

Specializing the generic micro-kernel in Listing 3 for ARM Neon can be achieved using the following six C macros:

```
// Macros for ARM Neon
#define vl_fp32 4

#define vregister float32x4_t
#define vinit()
#define vload(vreg, mem)  vreg = vld1q_f32(mem)
#define vstore(mem, vreg) vst1q_f32(mem, vreg)
#define vupdate(vreg1, vreg2, vreg3, j) \
                vreg1 = vfmaq_laneq_f32(vreg1, vreg2, vreg3, j)
```

Here, given that the vector registers in ARM Neon are 128-bit wide, we set `vl_fp32` to (128/32=) 4 (FP32 numbers).

A problem with ARM Neon is that the last argument of the intrinsic `vfmaq_laneq_f32` must be of type `const int`. This requires us to replace the innermost loop of the micro-tile update in the generic micro-kernel (Lines 34–35 in Listing 3) with the following fragment of code:

```
        // Original innermost loop of the micro-tile update
        // for ( j=0; j<vl_fp32; j++ )
        //   vupdate(Cr[iv][jv*vl_fp32+j], ar[iv], br[jv], j);

        // To be replaced for ARM Neon with the following.
        // Note that, for ARM_Neon, vl_fp32 = 4
        vupdate(Cr[iv][jv*vl_fp32+0], ar[iv], br[jv], 0);
        vupdate(Cr[iv][jv*vl_fp32+1], ar[iv], br[jv], 1);
        vupdate(Cr[iv][jv*vl_fp32+2], ar[iv], br[jv], 2);
        vupdate(Cr[iv][jv*vl_fp32+3], ar[iv], br[jv], 3);
```

This is equivalent to fully unrolling the innermost loop of the update, and it is straight-forward to carry out manually.

3.6 Adapting the generic micro-kernel to Intel AVX-512

Intel was a pioneer company to explore wide SIMD units, from 64 bits in MMX (Intel Pentium MMX2, 1997-1998), to 128 bits in SSE (Intel Pentium III, 1999), 256 bits in AVX (Intel Sandy Bridge, 2011), and 512-bit in AVX-512 (Intel Xeon Phi and Intel Skylake, 2013).

Adapting the generic micro-kernel to Intel AVX-512 is attained via the redefinition of the six C macros:

```
// Macros for Intel AVX-512
#define vl_fp32 16

#define vregister __m512
#define vinit()
#define vload(vreg, mem)  vreg = _mm512_loadu_ps(mem)
#define vstore(mem, vreg) _mm512_storeu_ps(mem, vreg)
#define vupdate(vreg1, vreg2, vreg3, j) \
                vreg1 = _mm512_fmadd_ps(vreg2, \
                        _mm512_set1_ps(vreg3[j]), \
                        vreg1)
```

Here we take into account that, for AVX-512, the vector registers are 512-bit long, and thus can contain vl_fp32=16 FP32 numbers.

Adapting the generic micro-kernel to Intel/AMD AVX2 presents minor differences and, therefore, it is omitted for brevity.

3.7 Adapting the generic micro-kernel to ARM SVE

ARM SVE (scalable vector extension) introduces a flexible intrinsics-based programming interface that supports variable-length SIMD units, of up to 2,048 bits.

In order to transform the generic micro-kernel with *C*-resident in Listing 3 into an SVE routine, we can define the following C macros:

```
// Macros for ARM SVE
#define vl_fp32 16

#define vregister svfloat32_t
#define vinit() svbool_t pred = svwhilelt_b32_u32(0, vl_fp32)
#define vload(vreg, mem)  vreg = svld1_f32(pred, mem)
#define vstore(mem, vreg) svst1_f32(pred, mem, vreg)
#define vupdate(vreg1, vreg2, mem, j) \
                vreg1 = svmla_n_f32_z(pred, vreg1, vreg2, mem)
```

In this particular example, we set vl_fp32 = 512/32 = 16 FP32 numbers to target an SVE-enabled processor with 512-bit vector registers. Also, we invoke the SVE intrinsic svwhilelt_b32_u32(0, vl_fp32) (via the macro vinit), to create an appropriate predicate (or mask) that operates with 512-bit vector registers.

In addition, we need to take into account the specific interface of the selected SVE instrinsic for the AXPY-like update. In particular, zsvmla_n_f32_z multiplies the contents of a vector with a scalar *that resides in memory* and then adds the result to the contents of a second vector. In contrast, the generic micro-kernel assumed that the axpy update performs the addition of two vectors, one of them scaled with a specific component of a third vector register. This requires two small changes in the generic realization to accommodate the SVE intrinsic:

1. The elements of *B* are no longer retrieved into vector registers using vector loads, but instead are retrieved one by one, as they are needed during the vector updates.

Therefore, the loop which loads B in the generic micro-kernel (Lines 28–29) should be removed in the ARM SVE version.

2. The vector update has to be modified to reflect the difference in the interface of the AXPY-like intrinsics:

```
        // Original operation of the micro-tile update
        // vupdate(Cr[iv][jv*vl_fp32+j], ar[iv], br[jv], j);

        // To be replaced for ARM SVE with the following:
        vupdate(Cr[iv][jv*vl_fp32+j], ar[iv],
                &Br[baseB+jv*vl_fp32+j], -1);
```

At this point we note that it is possible to produce a realization of the micro-kernel that is more similar to the original one using other SVE intrinsics. In particular, we can uploaded the row of B into vector registers, then broadcast each element into a separate vector register (via `svdup_lane_f32`), and finally update the micro-tile using `svmla_f32_z`. However, our experiments showed a lower performance due to the specific assembly instructions generated by the compiler for this option.

An additional hurdle for ARM SVE is that current compilers do not allow the declaration of arrays of vector registers. To avoid this, we created a Python (script) generator which, given a target pair (m_r, n_r), automatically produces the code for the generic micro-kernel in C, with all loops traversing these two dimensions unrolled and the arrays translated into scalar variables. Listing 4 shows an example of the output produced by the Python generator for an $mr \times n_r = 8 \times 4$ micro-kernel, with a vector register length `vl_fp32`=4.

```
// Macros for RISC-V V
#define vl_fp32 16

#define vregister vfloat32m4_t
#define vinit() \
          float init[vl_fp32]; \
          memset(&init, 0.0, vl_fp32); \
          vle32_v_f32m4(&init, vl_fp32))
#define vload(vreg, mem)  vreg = vle32_v_f32m4(mem, vl_fp32)
#define vstore(mem, vreg) vse32_v_f32m4(mem, vreg, vl_fp32)
#define vupdate(vreg1, vreg2, vreg3, j) \
        vreg1 = vfmacc_vf_f32m4(vreg1, vreg3[j], vreg2, \
                                 vl_fp32)
```

3.8 Adapting the generic micro-kernel to RISC-V V

RISC-V mimics SVE to define a collection of intrinsics that can accommodate SIMD units with variable-length. Assuming a platform with 512-bit SIMD units (that is, with `vl_fp32`=16), the specialization of the generic micro-kernel to employ the RISC-V Vector Extension 1.0 (RVV 1.0), is achieved via the following C macros:

```
void gemm_ukernel_Cresident_SIMD_8x4( int kc, float *Ar, float *Br,
                                      float *C,  int Clda ) {
// mr x nr = 8 x 4 micro-kernel with vl_fp32 = 4 and C resident in regs.
// Inputs:
//   - C  stored in column-major order, with leading dimension Clda
//   - Ar packed by columns, with leading dimension mr = 8
//   - Br packed by rows, with leading dimension nr = 4

  int       pr, baseA = 0, baseB = 0, mr = 8, nr = 4;
  vregister Cr00, Cr01, Cr02, Cr03,
            Cr10, Cr11, Cr12, Cr13; // Micro-tile of C
  vregister ar0, ar1, br0; // Single column/row of Ar/Br

  vinit();

  // Load the micro-tile of C into vector registers,
  // each with vl_fp32 = 4 FP32 elements
  vload(Cr00, &C[0*Clda+0]); vload(Cr01, &C[1*Clda+0]);
  vload(Cr02, &C[2*Clda+0]); vload(Cr03, &C[3*Clda+0]);
  vload(Cr10, &C[0*Clda+4]); vload(Cr11, &C[1*Clda+4]);
  vload(Cr12, &C[2*Clda+4]); vload(Cr13, &C[3*Clda+4]);

  for ( pr=0; pr<kc; pr++ ) {
    // Load the pr-th column/row of Ar/Br into vector registers
    vload(ar0, &Ar[baseA+0]); vload(ar1, &Ar[baseA+4]);
    vload(br0, &Br[baseB+0]);

    // Update the micro-tile with axpy
    vupdate(Cr00, ar0, br0, 0); vupdate(Cr01, ar0, br0, 1);
    vupdate(Cr02, ar0, br0, 2); vupdate(Cr03, ar0, br0, 3);
    vupdate(Cr10, ar1, br0, 0); vupdate(Cr11, ar1, br0, 1);
    vupdate(Cr12, ar1, br0, 2); vupdate(Cr13, ar1, br0, 3);
    baseA += mr; baseB += nr;
  }

  // Store the micro-tile in memory
  vstore(&C[0*Clda+0], Cr00); vstore(&C[1*Clda+0], Cr01);
  vstore(&C[2*Clda+0], Cr02); vstore(&C[3*Clda+0], Cr03);
  vstore(&C[0*Clda+4], Cr10); vstore(&C[1*Clda+4], Cr11);
  vstore(&C[2*Clda+4], Cr12); vstore(&C[3*Clda+4], Cr13);
}
```

Listing 4: Micro-kernel that operates with a 8×4 micro-tile and C-resident automatically obtained with the Python generator for a platform with `vl_fp32`=4.

4 Experimental evaluation

In this section, we assess the efficiency of the BLIS family of algorithms when combined with high-level micro-kernels in the context of DL applications.

4.1 Setup

We leverage three architectures in this section:

- An NVIDIA Carmel (ARMv8.2-based) processor embedded in the NVIDIA Jetson AGX Xavier board with 64 GiB of LPDDR4x memory. In order to reduce variability in the evaluation, the processor frequency in this platform was fixed to 2.3 GHz. The Operating System (OS) is Ubuntu 18.04, and we also utilized `gcc` v10.3, BLIS (version v0.8.1), OpenBLAS (version v0.3.19), and ARMPL (version v21.1).
- A Fujitsu A64FX processor (2.20 GHz) from a cluster at the Barcelona Supercomputing Center. This is equipped with 32 GiB of HBM2 memory, and runs a Red Hat Enterprise Linux 8 OS with the following software used for the experiments: `gcc` v10.2.0, and BLIS v0.8.1.
- An AMD EPYC 7282 16-core Processor (2.8 GHz), sited in a cluster at UPV, equipped with 504 GiB of DDR4 memory, and running a Ubuntu 10.04.6 LTS OS (Linux Kernel 4.15.0) with the following software: `gcc` v8.4.0, and BLIS v0.9.0.[1]

A single core is employed in the three architectures, with a thread bound to it. (The multi-threaded, loop-level parallelism of the BLIS family of algorithms has been analyzed elsewhere [13, 14].) All experiments are carried out in IEEE FP32 arithmetic, and they are repeated a large number of times, reporting the average results in this section. Performance is measured in terms of billions of floating point operations per second, abbreviated as GFLOPS.

Given the extreme interest in deploying DL technologies, the dataset for the experimentation includes matrix mutiplications with their dimensions determined by the application of the IM2COL-approach to the convolution layers in the neural networks ResNet-50 v1.5 [15], VGG16 [16] and GoogleLeNet [17], combined with the ImageNet dataset. The batch size is set to 1 sample, reflecting latency-oriented scenario [8, 9]. In the following experiments, we focus on the performance gains that can be obtained when leveraging specific micro-kernels for the convolution operators in these neural networks. The global impact of these gains on the complete inference process depends on external factors to our work, such as the level of optimizations of other components. For example, in [9] we report that the convolution layers in the ResNet-50 v1.5 model (combined with ImageNet) can consume between 45% and 87% of the inference time, depending on the optimizations that were applied.

4.2 ARM Neon on NVIDIA Carmel

Figure 5 shows the performance obtained for (the matrix multiplications resulting from the application of IM2COL to) two convolution layers of Resnet-50 v1.5. For clarify, we only report results for the B3A2C0 algorithm for GEMM, combined with 14 different micro-kernels, but omit the combinations for the other five variants of the BLIS family. We also refrain from evaluating the micro-kernels which require a number of vector registers that exceeds those available in the hardware (32 in the NVIDIA Carmel processor). For reference, we also include the performance

[1] The AOCL library from AMD is actually a compilation of BLIS with the proper configuration flags, that we have used in our evaluation.

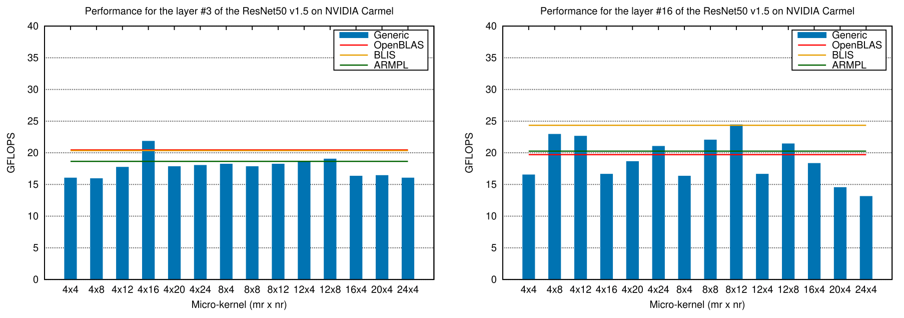


Fig. 5 Performance of the B3A2C0 algorithm with different micro-kernels for the convolutional layers #3 and #16 in Resnet-50 v1.5 (left and right, respectively) on NVIDIA Carmel

attained with the implementations of GEMM in BLIS, OpenBLAS and ARMPL specifically tuned for this processor. This experiment demonstrates the benefits of leveraging different micro-kernels for the GEMM operation instead of the single kernel approach that conventional libraries offer. (For example, BLIS only offers the baseline algorithm combined with a micro-kernel of size 8×12 for the NVIDIA Carmel processor.) In this scenario, we can benefit from the 4×16 micro-kernel in layer #3 and from the 8×12 micro-kernel for the layer #16. In both layers, with the best micro-kernel choice, the generic implementation outperforms the other three fine-tuned GEMM implementations by a non-negligible margin.

Figure 6 reports the performance of the different implementations of GEMM for all the convolutional layers of Resnet-50 v1.5, VGG16 and GoogleLeNet on the NVIDIA Carmel processor. In this experiment, the results of the generic approach correspond to the best algorithm and micro-kernel for each layer. For this purpose, we evaluated all six algorithms, combined with micro-kernels of dimensions 4×8, 4×12, 4×16, 4×20, 4×24, 8×12, 12×4, 12×8, 16×4, 20×4, and 24×4.

For the Resnet-50 v1.5 model (top-left plot), OpenBLAS is the best option for three layers and BLIS for five layers. In comparison, the flexibility of the generic algorithm results in this being the option in 12 out of the 20 layers. For the VGG16 model (top-right plot), the generic algorithm outperforms the other libraries in four out of nine cases; BLIS is the best choice for four layers; and ARMPL is optimal for only one layer. Finally, for GoogleLeNet (bottom two plots), the generic algorithm offers the best performance in 41 out of 53 layers; OpenBLAS is the best option for layer #1 only; ARMPL is preferable for two layers; and BLIS for six layers.

4.3 ARM SVE on Fujitsu A64FX

Figure 7 reports the results from the analysis on the Fujitsu A64FX processor. In this case, we compare our approach with the implementation of BLIS only as we have no access to optimized instances of OpenBLAS and ARMPL for this platform. For the generic algorithm, we have implemented and analyzed six micro-kernels: 32×10, 32×12, 32×14, 48×8, 64×6, and 80×4. In this platform, the BLIS library is extremely hand-tuned and includes aggressive prefetching techniques

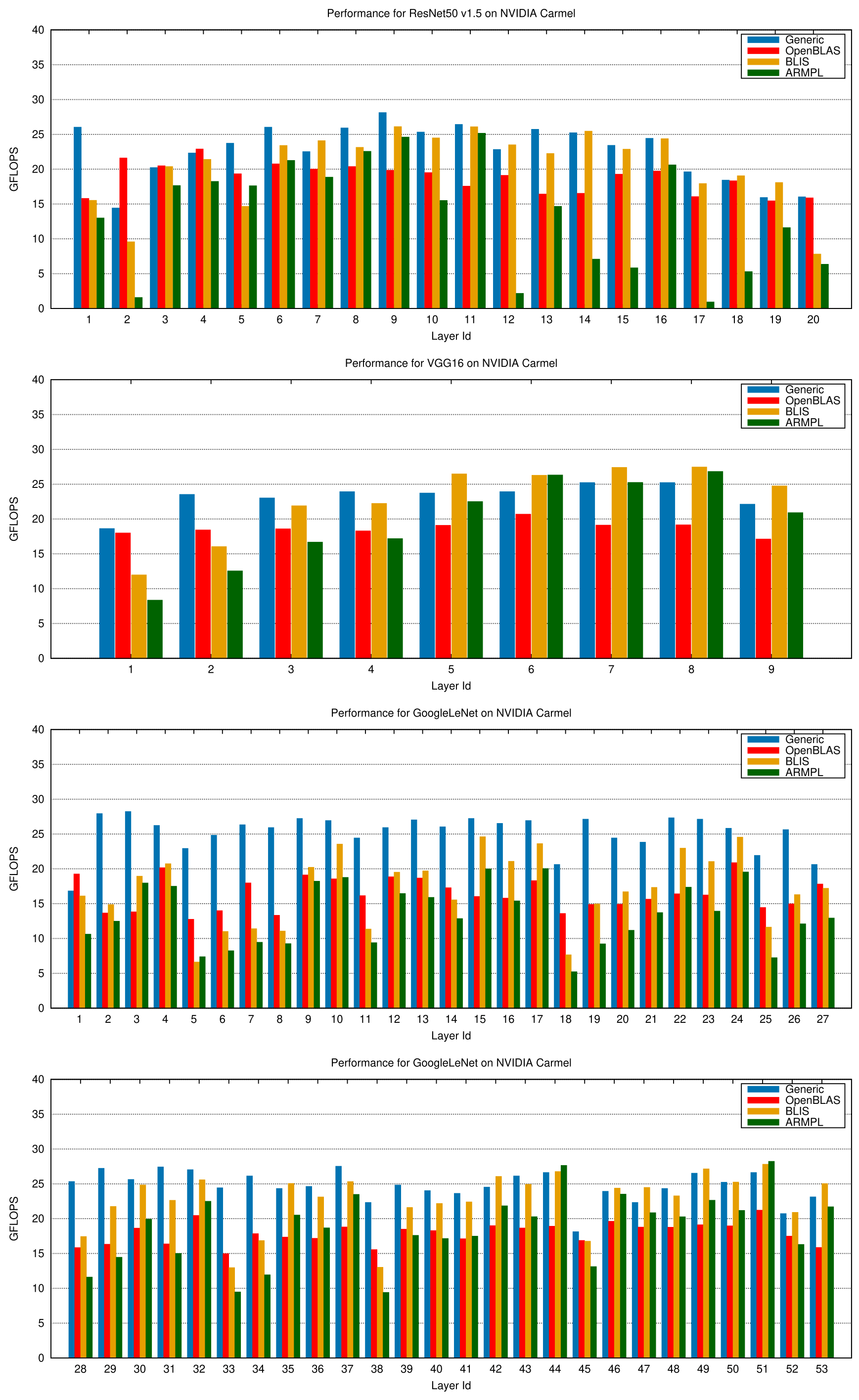


Fig. 6 Performance of the best algorithm+micro-kernel on NVIDIA Carmel

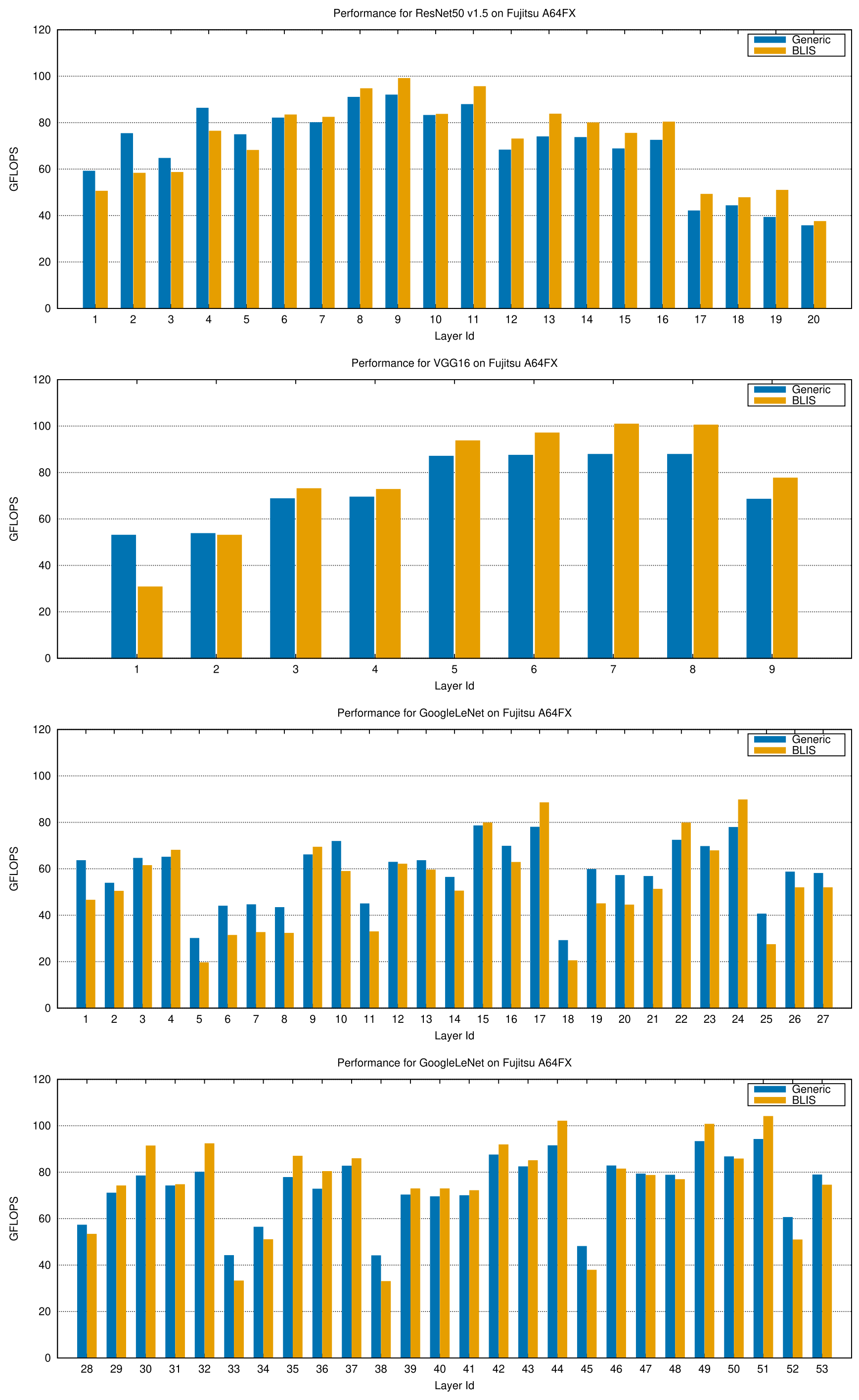


Fig. 7 Performance of the best algorithm+micro-kernel on Fujitsu A64FX

which result in an extra burst of performance. In comparison the micro-kernels integrated in the generic implementation do not leverage this technique. As a result, in the Resnet-50 v1.5 and VGG16 models (top-left and top-right plots, respectively), the generic algorithm is only competitive in about 20% of the layers. However, this occurs for the very first layers, which concentrate the most time-consuming operations. In contrast, for the GoogleLeNet model (bottom two plots), the generic algorithm outperforms BLIS in 32 of 53 layers. In summary, as was the case for the Carmel processor, the flexibility of generic micro-kernels selection benefits the overall performance of the GEMM operations.

At this point, we note that some of the prefetching techniques featured by the BLIS routine can be also integrated into the intrinsics-based micro-kernel. However, this points in the direction of an architecture-specific solution, thus, is against the portability flag raised by this work. For this reason, exploring this option is left as part of future work.

4.4 AMD EPYC 7282

Figure 8 shows the results from the same round of experiments, this time carried out on the AMD EPYC 7282 processor. For reference, in this case we only include in the comparison BLIS as AMD's native library (AOCL) is just a version of BLIS in disguise. Given that the AMD EPYC 7282 supports AVX2 (256-bit SIMD units) and features 16 256-bit SIMD registers, we developed and tested micro-kernels of dimensions 16×6, 24×4. In addition, we mimicked AOCL-BLIS to develop two additional micro-kernels, of dimensions 6×16 and 4×6, that operate with matrices stored by rows and avoid packing the entries of A into the buffer A_c for the baseline algorithm B3A2C0. The results in the figure show that our implementation for GEMM delivers a GFLOPS rate that is comparable in most cases to that obtained with the BLIS realization, showing relevant benefits for some special layers but also loosing by a non-negligible margin in a few others.

4.5 Discussion

To close this section, we link the ultimate reasons which determine the performance of the different micro-kernels with the analytical model in [10]. For this purpose, we expose the relationship using as a workhorse the BLIS baseline algorithm (B3A2C0), the NVIDIA Carmel processor, and layer #1 of ResNet50 v1.5. For that layer, the dimensions of the GEMM are given by the following tuple: $(m, n, k) = (12544, 64, 147)$.

For that particular processor/layer, the analytical model in [10] reports that, due to the reduced k-dimension of the problem, the micro-panel of B_c that targets the L1 cache only occupies 10.8% of that memory level on the NVIDIA Carmel for the $m_r \times n_r = 8 \times 12$ micro-kernel that is integrated BLIS. (In theory, this micro-kernel should have occupied up to 50% with that micro-panel of B_c, reserving the rest of the L1 cache for entries from the A, B operands). An additional problem arises for the L2 cache: With the cache configuration parameters fixed in BLIS to

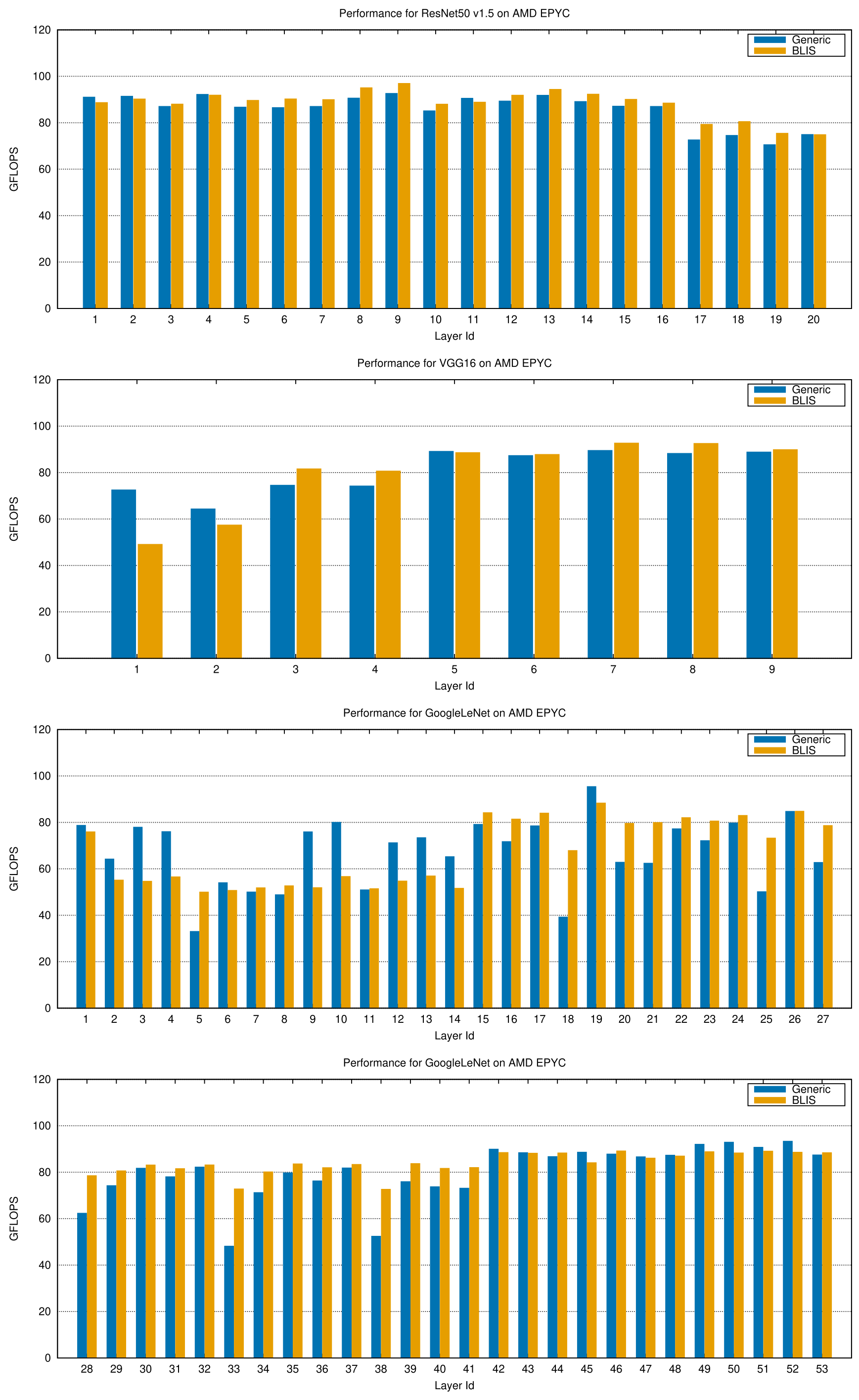


Fig. 8 Performance of the best algorithm+micro-kernel on AMD EPYC 7282

$m_c = 896, k_c = 512$ for the NVIDIA Carmel, the small m-dimension of the problem yields that the buffer A_c only occupies 25.1% of the L2 cache. (In theory, for the 8×12 micro-kernel, the buffer A_c should have occupied up to 81.2% of that memory level leaving the rest for entries of C, B.)

Let us next consider a micro-kernel of dimension $m_r \times n_r = 4 \times 24$. (The optimal micro-kernel among those that we evaluated for this layer.) In this case, the occupancy of the L1 cache by the micro-panel of B_c grows to 21.5%, which is clearly superior to that observed for the (BLIS) 8×12 micro-kernel. In addition, for the 4×24 micro-kernel, the utilization of the L2 cache by the buffer A_c is 75.0% which, according to the analytical model, is the maximum that should be dedicated to this operand (in each case).

In summary, there is a clear theoretical benefit from adopting an $m_r \times n_r = 4 \times 24$ micro-kernel for this particular case, which is conformal with the performance advantage that is reported in Fig. 6 when using the "Generic" (4×24) micro-kernel versus BLIS.

5 Concluding remarks and future work

Building efficient code for a given computational kernel with minimal effort has been always a great challenge. With each new architecture, it becomes necessary to revisit the kernel in order to obtain an optimized version. This is the case, for example, of numerical linear algebra libraries in general, and of matrix multiplication in particular.

The advent of deep neural networks together with an explosion in the variety of computing devices to run DL applications, especially for the inference phase, require a significant step aimed at finding optimal methodologies to produce efficient matrix multiplication codes. In particular, depending on the target neural model layer, the dimensions of the intervening matrices are in general different; consequently, so is the performance of a general matrix multiplication kernel. Therefore, it is necessary to design a collection of computational kernels optimized for the matrix product, not only for the underlying architecture, but also for each dimension of the operands.

The use of intrinsics greatly facilitates vector programming since it allows to work at a high level, leaving in the hands of the compiler the translation-to-assembler and other optimization tasks which are difficult for the programmer. However, the different ISAs (instruction set architectures) still complicate optimizing the code for the wide range of processors within reach.

Given all of the above, combining all the necessary optimizations specific to each architecture and computational kernel in a single code is a difficult challenge. To tackle with this issue, in this work we have proposed a unique generic code for several (very different) architectures which are spread use nowadays. This generic routine, implemented in C together with a reduced set of particular macros for each processor type, allows generating optimized code based on vector intrinsics, to obtain a varied set of micro-kernels on which a matrix multiplication operation is based.

The experimental results, obtained with two ARM-based processor architectures, offer some optimism about the future of the proposed solution. Although we have not achieved the best performance with the generic kernel in all cases, the reduced implementation effort required for this purpose leads us to conclude that the trade-off is favorable to the proposed solution.

Acknowledgements We thank the Barcelona Supercomputing Center for granting the access to the Fujitsu A64FX system were the developments and tests were performed.

Author contributions AC implemented the micro-kernel generator script, wrote the results section of the paper, and mentored GA who executed the Carmel processor experiments. GA also generated the Figures for this architecture. HM, implemented the micro-kernels for the Fujitsu A64FX and AMD Rome platforms, executed the tests over these devices, and generated their Figures. PA-J and FDI wrote several sections of the paper as well as conducted distinct research steps. ESQ-O guided the overall research, wrote different sections, and revised the entire paper.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by the research projects PID2020-113656RB-C22, RTI2018-093684-B-I00, and PID2021-126576NB-I00, of MCIN/AEI/10.13039/501100011033 and by “ERDF A way of making Europe”, and CM via Multiannual Agreement with Complutense University in the line Program to Stimulate Research for Young Doctors in the context of the V PRICIT under projects PR65/19-22445 and CM S2018/TCS-4423. A. Castelló is a FJC2019-039222-I fellow supported by MCIN/AEI/10.13039/501100011033. H. Martínez is a postdoctoral fellow supported by the *Consejería de Transformación Económica, Industria, Conocimiento y Universidades de la Junta de Andalucía*.

Data availibility BLIS is available at https://github.com/flame/blis OpenBLAS is available at https://github.com/xianyi/OpenBLAS ARMPL is available at https://developer.arm.com/downloads/-/arm-performance-libraries

Declarations

Conflict of interest The authors declare that they have no competing interests.

References

1. Dongarra JJ, Du Croz J, Hammarling S, Duff I (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1):1–17
2. Goto K, van de Geijn RA (2008) Anatomy of a high-performance matrix multiplication. ACM Trans Math Softw 34(3):12:1-12:25
3. Van Zee FG, van de Geijn RA (2015) BLIS: a framework for rapidly instantiating BLAS functionality. ACM Trans Math Softw 41(3):14:1-14:33
4. Xianyi Z, Qian W, Yunquan Z (2012) Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)
5. Smith TM, van de Geijn RA (2019) The MOMMS family of matrix multiplication algorithms. CoRR, vol. abs/1904.05717. [Online]. Available: http://arxiv.org/abs/1904.05717

6. Gunnels JA, Gustavson FG, Henry GM, van de Geijn RA (2004) A family of high-performance matrix multiplication algorithms. In: Proc. 7th Int. Conf. on Applied Parallel Computing: State of the Art in Scientific Computing, ser. PARA'04, pp 256-265
7. Castelló A, Igual FD, Quintana-Ortí ES (2022) Anatomy of the BLIS family of algorithms for matrix multiplication. In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp 92–99
8. Chellapilla K, Puri S, Simard P (2006) High performance convolutional neural networks for document processing. In: International Workshop on Frontiers in Handwriting Recognition
9. Barrachina S, Dolz MF, San Juan P, Quintana-Ortí ES (2022) Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors. J Parallel Distrib Comput 167(C):240–254
10. Low TM, Igual FD, Smith TM, Quintana-Ortí ES (2016) Analytical modeling is enough for high-performance BLIS. ACM Trans Math Softw 43(2):12:1-12:18
11. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76
12. Dowd K, Severance CR (1998) High performance computing, 2nd ed. O'Reilly
13. Zee FGV, Smith TM, Marker B, Low TM, Geijn RAVD, Igual FD, Smelyanskiy M, Zhang X, Kistler M, Austel V, Gunnels JA, Killough L (2016) The BLIS framework: experiments in portability. ACM Trans Math Softw 42(2):1–19
14. Smith TM, van de Geijn R, Smelyanskiy M, Hammond JR, Zee FGV (2014) Anatomy of high-performance many-threaded matrix multiplication. In: Proc. IEEE 28th Int. Parallel and Distributed Processing Symp. ser. IPDPS'14, pp 1049–1059
15. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 770–778
16. Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556
17. Szegedy C, et al. (2014) Going deeper with convolutions, CoRR, vol. abs/1409.4842, [Online]. Available: http://arxiv.org/abs/1409.4842

Authors and Affiliations

Guillermo Alaejos[1] · Adrián Castelló[1] · Héctor Martínez[2] · Pedro Alonso-Jordá[1] · Francisco D. Igual[3] · Enrique S. Quintana-Ortí[1]

Guillermo Alaejos
galalop@upv.es

Héctor Martínez
el2mapeh@uco.es

Pedro Alonso-Jordá
palonso@upv.es

Francisco D. Igual
figual@ucm.es

Enrique S. Quintana-Ortí
quintana@disca.upv.es

1 Universitat Politècnica de València, València, Spain

2 Universidad de Córdoba, Córdoba, Spain

3 Universidad Complutense de Madrid, Madrid, Spain